https://en.wikibooks.org/w/index.php?title=Artificial\_Intelligence/Definition&stable=0#Uninformed\_Search

DFS

[**Depth-First-Search**](https://en.wikipedia.org/wiki/Depth-first_search)

* Explores one path to the deepest level and then backtracks until it finds a goal state. This is realized in a *LIFO* queue (i.e. stack).
* **DFS** is *complete* (if the search tree is finite) and *incomplete* (if the search tree is infinite)
* It is **not** *optimal* (it stops at the first goal state it finds, no matter if there is another goal state that is shallower than that).
* Space: {\displaystyle O(bm)} (much lower than **BFS**).
* Time: {\displaystyle O(b^{m})} (Higher than **BFS** if there is a solution on a level smaller than the maximum depth of the tree).
* Danger of running out of memory or running indefinitely for infinite trees.

**Time Complexity:** If you can access each node in O(1) time, then with branching factor of b and max depth of m, the total number of nodes in this tree would be = b \* b \*b … m times = bm, resulting in total time to visit each node proportional to bm. Hence the complexity = O(bm).

On the other hand, if instead of using the branching factor and max depth you are given the number of nodes *n* directly, you can say that the complexity will be proportional to *n* or equal to *O(n)*.

The other answers of SO that you have linked in your question are similarly using different terminologies. The idea is same everywhere. Some answers have added the edge count too in order to make the answer more precise but in general, node count is sufficient to describe the complexity.

**Space Complexity:** The length of longest path = m. For each node you have **to store its siblings** so that when you are done visiting all the children and you come back to a parent node, you can know which sibling to explore next. For m nodes down the path, you will have to store b nodes extra for each of the m nodes. That’s how you get a O(bm) space complexity.

DFS(G,v) ( v is the vertex where the search starts )

Stack S := {}; ( start with an empty stack )

for each vertex u, set visited[u] := false;

push S, v;

while (S is not empty) do

u := pop S;

if (not visited[u]) then

visited[u] := true;

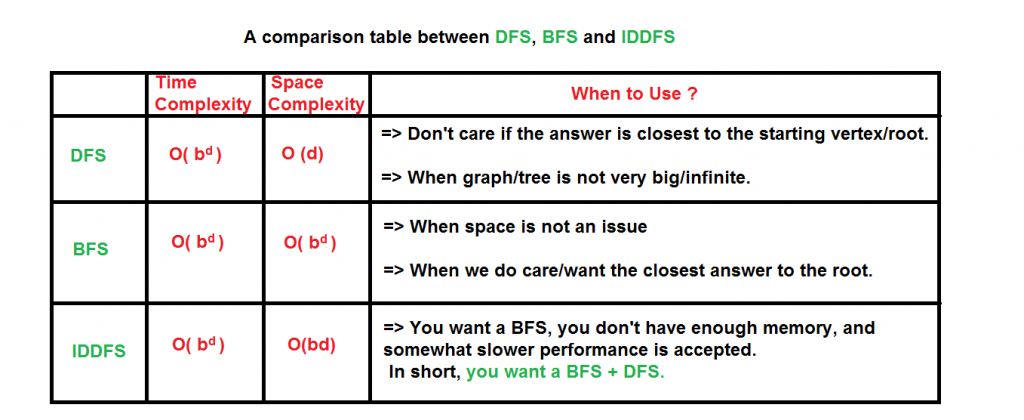
for each unvisited neighbour w of u

push S, w;

end if

end while

END DFS()



**Advantages of  BFS:-**  
  
1. Solution will definitely found out by BFS If there are some solution.  
2. BFS will never get trapped in blind alley , means unwanted nodes.  
3. If there are more than one solution then it will find solution with minimal steps.  
  
**Disadvantages Of BFS :**-  
1. Memory Constraints As it stores all the nodes of present level to go for next level.  
2. If solution is far away then it consumes time.  
  
**Application Of BFS :**-  
1.Finding Shortest Path.  
2.Checking graph with bipertiteness.  
3.Copying cheiney's Algorithm.  
  
**Advantages Of DFS** :-  
1. Memory requirement is Linear WRT Nodes.  
2. Less time and space complexity rather than BFS.  
3. Solution can be found out by without much more search.  
  
**Disadvantage of DFS :-**  
1. Not Guaranteed that it will give you solution.  
2. Cut-off depth is smaller so time complexity is more.  
3. Determination of depth until the search has proceeds.  
  
**Applications of DFS:-**   
1. Finding Connected components.  
2. Topological sorting.  
3. Finding Bridges of graph.

+ - DFS AND BFS -http://cgi.csc.liv.ac.uk/~alatham/COMP219/COMP219%20Lec6%20searchStrategies%20notes.pdf

Examples - <https://web.uvic.ca/~maryam/AISpring94/Slides/02_UninfoSearch.pdf> https://www.ics.uci.edu/~welling/teaching/271fall09/HW3\_sol.pdf

## BFS CODE

https://tutorialedge.net/artificial-intelligence/breadth-first-search-java/

**Informed Search**[[edit](https://en.wikibooks.org/w/index.php?title=Artificial_Intelligence/Definition&action=edit&section=14" \o "Edit section: Informed Search)]

In *informed* search, a [heuristic](https://en.wikipedia.org/wiki/Heuristic_(computer_science)) is used as a guide that leads to better overall performance in getting to the goal state. Instead of exploring the search tree blindly, one node at a time, the nodes that we could go to are ordered according to some [evaluation function](https://en.wikipedia.org/wiki/evaluation_function) {\displaystyle h(n)} that determines which node is probably the "best" to go to next. This node is then expanded and the process is repeated (i.e. [**Best First Search**](https://en.wikipedia.org/wiki/Best_First_Search)). [**A\* Search**](https://en.wikipedia.org/wiki/A-star_search_algorithm) is a form of **BestFS**. In order to direct the search towards the goal, the [evaluation function](https://en.wikipedia.org/wiki/evaluation_function) must include some estimate of the cost to reach the closest goal state from a given state. This can be based on knowledge about the problem domain, the description of the current node, the search cost up to the current node **BestFS** optimizes **DFS** by expanding the most promising node first. Efficient selection of the current best candidate is realized by a [priority queue](https://en.wikipedia.org/wiki/priority_queue).

1. **Greedy Search**:
   * Minimizes the estimated cost to reach the goal. The node that is closest to the goal according to {\displaystyle h(n)} is always expanded first. It optimizes the search locally, but not always finds the global optimum.
   * It is not complete (can go down an infinite branch of the tree).
   * It is not optimal.
   * Space: {\displaystyle O(b^{m})} for the worst case.
   * Same for time, but can be reduced by choosing a good heuristic function.
2. [**A\* Search**](https://en.wikipedia.org/wiki/A-star_search_algorithm):
   * Combines *uniform cost search* (i.e. expand node on path with lowest cost so far) and **greedy search**. Evaluation function is {\displaystyle f(n)=g(n)+h(n)} (or estimated cost of the cheapest solution through node n). It can be proven that A\* is complete and optimal if {\displaystyle h} is [admissible](https://en.wikipedia.org/wiki/A_star#Intuition_about_why_A.2A_is_admissible_and_computationally_optimal) - that is, if it *never overestimates* the cost to reach the goal. This is optimistic, since they think the cost of solving the problem is less than it actually is.
   * Examples for {\displaystyle h}:
     + Path-Finding in a map: Straight-Line-Distance.
     + 8-Puzzle: Manhattan-Distance to Goal State.
     + Everything works, it just has to be admissible (e.g. {\displaystyle h(n)=0} always works, but transforms A\* back to uniform cost search).
   * If a [heuristic function](https://en.wikipedia.org/wiki/Heuristic_evaluation) {\displaystyle h\_{1}} estimates the actual distance to the goal better than another heuristic function {\displaystyle h\_{2}}, then {\displaystyle h\_{1}} *dominates* {\displaystyle h\_{2}}.
   * A\* maintains an open list ([priority queue](https://en.wikipedia.org/wiki/priority_queue)) and a closed list (visited nodes). If a node is expanded that's already in the closed list, stored with a lower cost, the new node is ignored. If it was stored with a higher cost, it is deleted from the closed list and the new node is processed.
   * {\displaystyle h} is **monotonic**, if the difference between the heuristics of any two connected nodes does not overestimate the actual distance between those nodes. Example of a **non-monotonic** heuristic: {\displaystyle n} and {\displaystyle n'} are two connected nodes, where {\displaystyle n} is the parent of {\displaystyle n'}. Suppose {\displaystyle g(n)=3} and {\displaystyle h(n)=4}, then we know that the true cost of a solution path through {\displaystyle n} is at least 7. Now suppose that {\displaystyle g(n')=4} and {\displaystyle h(n')=2}. Hence, {\displaystyle f(n')=6}.
     + First off, the difference in the heuristics (that is, 2) overestimates the actual cost between those nodes (which is 1).
     + However, we know that any path through {\displaystyle n'} is also a path through {\displaystyle n} and we already know that any path through {\displaystyle n} has a true cost of at least 7. Thus, the f-value of 6 for {\displaystyle n'} is meaningless and we will consider its parent's f-value.
   * {\displaystyle h} is consistent, if the h-value for a given node is less or equal than the actual cost from this node to the next node plus the h-value from this next node (triangular inequality).
   * If {\displaystyle h} is **admissible** and **monotonic**, the first solution found by A\* is guaranteed to be the optimal solution (open/close list bookkeeping is no longer needed).
3. **Summary**  
   So when to use DFS over A\*, when to use Dijkstra over A\* to find the shortest paths ?  
   We can summarise this as below-
4. 1) One source and One Destination-  
   → Use A\* Search Algorithm (For Unweighted as well as Weighted Graphs)
5. 2) One Source, All Destination –  
   → Use BFS (For Unweighted Graphs)  
   → Use Dijkstra (For Weighted Graphs without negative weights)  
   → Use Bellman Ford (For Weighted Graphs with negative weights)
6. 3) Between every pair of nodes-  
   → Floyd-Warshall  
   → Johnson’s Algorithm

n numerical analysis, **hill climbing** is a [mathematical optimization](https://en.wikipedia.org/wiki/Optimization_(mathematics)) technique which belongs to the family of [local search](https://en.wikipedia.org/wiki/Local_search_(optimization)). It is an [iterative algorithm](https://en.wikipedia.org/wiki/Iterative_algorithm) that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an [incremental](https://en.wikipedia.org/wiki/Incremental_heuristic_search" \o "Incremental heuristic search)change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.\

Това е итеративен алгоритъм, който започва с произволно решение на проблем, след което се опитва да намери по-добро решение, като направи постепенна промяна на решението. Ако промяната създаде по-добро решение, се прави друга постепенна промяна на новото решение и така нататък, докато не бъдат намерени допълнителни подобрения

Идеята за бавното охлаждане е внедрена в алгоритъма на симулираното закаляване като бавно намаляване на вероятността за приемане на по-лоши [кандидат-решения](https://bg.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D0%B4%D0%B8%D0%B4%D0%B0%D1%82-%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5" \o "Кандидат-решение)при претърсването на пространството на решенията. Приемането на по-лоши решения е фундаментално свойство на метаевристичните методи, тъй като то позволява по-изчерпателното претърсване на пространството на оптимално решение и помага за избягването на „забиването“ в локални оптимуми.

# Beam search

n [computer science](https://en.wikipedia.org/wiki/Computer_science), **beam search** is a [heuristic](https://en.wikipedia.org/wiki/Heuristic_(computer_science)) [search algorithm](https://en.wikipedia.org/wiki/Search_algorithm) that explores a graph by expanding the most promising node in a limited set. Beam search is an optimization of [best-first search](https://en.wikipedia.org/wiki/Best-first_search) that reduces its memory requirements. Best-first search is a graph search which orders all partial solutions (states) according to some heuristic. But in beam search, only a predetermined number of best partial solutions are kept as candidates.[[1]](https://en.wikipedia.org/wiki/Beam_search#cite_note-1) It is thus a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm).

## Details[[edit](https://en.wikipedia.org/w/index.php?title=Beam_search&action=edit&section=1" \o "Edit section: Details)]

Beam search uses [breadth-first search](https://en.wikipedia.org/wiki/Breadth-first_search) to build its [search tree](https://en.wikipedia.org/wiki/Tree_traversal). At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of heuristic cost.[[3]](https://en.wikipedia.org/wiki/Beam_search#cite_note-3) However, it only stores a predetermined number, {\displaystyle \beta }, of best states at each level (called the beam width). Only those states are expanded next. The greater the beam width, the fewer states are pruned. With an infinite beam width, no states are pruned and beam search is identical to [breadth-first search](https://en.wikipedia.org/wiki/Breadth-first_search). The beam width bounds the memory required to perform the search. Since a goal state could potentially be pruned, beam search sacrifices completeness (the guarantee that an algorithm will terminate with a solution, if one exists). Beam search is not optimal (that is, there is no guarantee that it will find the best solution).

In general, beam search returns the first solution found. Beam search for [machine translation](https://en.wikipedia.org/wiki/Machine_translation) is a different case: once reaching the configured maximum search depth (i.e. translation length), the algorithm will evaluate the solutions found during search at various depths and return the best one (the one with the highest probability).

The beam width can either be fixed or variable. One approach that uses a variable beam width starts with the width at a minimum. If no solution is found, the beam is widened and the procedure is repeated.[[4](https://en.wikipedia.org/wiki/Beam_search#cite_note-4)

# GENETIC ALGORITHMS

A simple hill-climber will explore one (local) optimum. What would you prefer -- one choice or many (but random) choices? No matter how bad there other options are, what would you lose to consider them? Thus, even random choices are better than no choices. In addition to having more choices, evolution is tied to "survival of the fittest" -- all evolutionary algorithms have a bias to the best of the available (random) choices. Thus, despite having randomness/stochasticness, these biases invariably lead evolutionary algorithms to be better than random.

If you have perfect knowledge, use it. If not, do some exploration to expand your knowledge. If this exploration has to be random, so be it.

Genetic Algorithms are not chaotic, they are stochastic. The complexity depends on the genetic operators, their implementation (which may have a very significant effect on overall complexity), the representation of the individuals and the population, and obviously on the fitness function. Given the usual choices (point mutation, one point crossover, roulette wheel selection) a Genetic Algorithms complexity is O(g(nm + nm + n)) with g the number of generations, n the population size and m the size of the individuals. Therefore the complexity is on the order of O(gnm)).  
This is of cause ignoring the fitness function, which depends on the application.

Like Bashar said, the answer to your question mainly depends on the fitness function you try to optimize. If it's a simple function like Rastrigin, you get O(NL) to evaluate the whole population. Otherwise, the complexity can be of any kind and is problem dependent.

The rest depends on the genetic operators you want to use.

For example, a stochastic selection will require you to sort the population so you end up with O(N log N). Tournament selection will be O(N).

Transforming the population with crossover and mutation operator will usually take O(NL).

To sum up, the anwser to your question will usually depend on (in order of significance):

1. The fitness function

2. The selection operator

3. The variation operators